Java* API Support for Profiling With VTune

Background

VTune, Intel's Visual Tuning Environment, features a low-overhead, non-intrusive profiler. VTune has been very successful in helping performance-sensitive ISVs write applications for the Pentium®, Pentium Pro, and Pentium II processors, and the Pentium processor with MMX™ technology. VTune monitors the entire system, giving a "bird's eye" view of the system performance, with the capability of zooming into the source line/assembly instruction of the module in which the user is interested. VTune also uses performance simulators for various Intel processors to give detailed information about critical sections of the user code.

VTune monitors the system based on the time-based or event-based sampling methods. In time-based sampling, VTune's device driver records the PC (CS/EIP) values periodically (based on fixed time intervals) and, after the sampling period, analyzes the data to associate the samples with the actual modules running in the system. Event-based sampling is similar, but the device driver records the PC after a fixed number of processor events (such as cache misses) have occurred. This allows the user to see where exactly the highest frequency of processor events occurred in the code. After the sampling and analysis, VTune breaks down the executable when the user zooms into it, and performs a static code analysis to see how the code performs on various Intel processors, and associates the samples (if any) to the code. VTune then "completes" the tuning process by giving tuning advice in the form of source code transformations to improve the performance.

Java* and VTune

VTune's identification of the appropriate module (using PC values) is possible due to the fact that VTune knows where the OS and user mode modules are loaded (it also gets notified when they are unloaded). VTune knows at any given time to which module a particular memory address belongs. With the introduction of the Java* VM and JIT, the picture changes slightly. In Java, methods can be interpreted, JITed, or natively compiled.

When a Java method is JITed and executed, the code address that is executed is a dynamically created one; therefore, VTune will not be able to associate this with any module. In addition, when a Java method is interpreted, the time will be associated with the interpreter and not the Java method.

Thus, in its current form, when VTune profiles a Java application, the time is either associated to **JAVAi.dll** (when a method is interpreted) or to "OTHER32", a bucket for all 32-bit PC samples that VTune cannot associate with a module (when a method is JITed). For a Java application developer (and for JIT writers), it is important that the time be associated with the actual "Java method."

The API Proposal

We are currently working with various software vendors to provide support for Java profiling with VTune. The basic support VTune needs is the following:

1. VTune needs to know when a class is loaded and unloaded.
2. VTune needs to know when a method is JITed (before starting the execution), the method and the class names, the address where the code was generated, and the size of the code.
3. VTune needs symbol information on the JITed code: the line number to code mappings, symbols (like procedure and variable names) to address mappings, and, if possible, byte code-to-native code mappings.
4. VM/JIT should be able to communicate with a profiler object (possibly a DLL) and notify the object of the above mentioned events and provide the necessary information. The profiler object, in turn, will communicate with rest of VTune. The JIT/VM can be “enabled” for profiling through some registry settings.

VTune Support Levels

VTune support can be provided at different levels by JIT and VM vendors. Depending on the level of support available, VTune can provide different levels of information to the users on the Java program performance. The following paragraphs outline varying levels of support:

1. ***Minimum Support in JIT***:

A JIT vendor can provide a minimum set of information to VTune to help in the performance analysis. In this method, a simple profile DLL is made available to the JIT writers. JIT loads the profile DLL based on its availability, and JIT can make calls to the profile DLL whenever a method is JITed, with the following set of definitions:

This minimum level of support does not include a JVM/JIT API for enumeration of currently loaded methods. If there is no API to find out what is currently loaded, then the profile DLL must be loaded at the start of the JVM, and it is the responsibility of the profile DLL to maintain a list of currently loaded JITed methods.

The file is “iJITProf.h”

```
#ifndef __iJITProf_h__
#define __iJITProf_h__

#include "windows.h"

#ifdef __cplusplus
extern "C"{
#endif

 typedef enum iJIT_jvm_event
 {
   // execution
        JVM_EVENT_TYPE_EXCEPTION_OCCURRED,    // An exception occurred.  The exception
                                               // handler will be executed in StackID.

   // object monitors
        JVM_EVENT_TYPE_MONITOR_BLOCKED,       // The current thread blocked while trying
                                               // to acquire an ObjectID’s object monitor.
        JVM_EVENT_TYPE_MONITOR_ACQUIRED,      // The current thread acquired ObjectID’s
```

```
                                                        // object monitor.
    JVM_EVENT_TYPE_MONITOR_RELEASED,      // The current thread released ObjectID's
                                                        // object monitor.

  // threads
   JVM_EVENT_TYPE_THREAD_CREATE,        // A Java thread ThreadID is being created.
  JVM_EVENT_TYPE_THREAD_DESTROY,       // A Java thread ThreadID is being destroyed.

  // class loading
  JVM_EVENT_TYPE_CLASS_LOAD_STARTED,    // A class ClassID is being loaded.
  JVM_EVENT_TYPE_CLASS_LOAD_FINISHED,   // A class ClassID has been loaded.

  // garbage collection
  JVM_EVENT_TYPE_GC_STARTED,           // The garbage collector started.
  JVM_EVENT_TYPE_GC_FINISHED,          // The garbage collector completed.

  // shutdown
  JVM_EVENT_TYPE_SHUTDOWN_NORMAL,       // Program exiting normally.
  JVM_EVENT_TYPE_SHUTDOWN_ERROR,        // Program or Java VM encountered a fatal error.
  JVM_EVENT_TYPE_SHUTDOWN_INTERRUPTED,  // Program or Java VM interrupted (control-C).

  // For VTune
  JVM_EVENT_TYPE_METHOD_LOAD_FINISHED,  // issued after method code jitted
                                          // into memory but before code is
                                          // executed

  JVM_EVENT_TYPE_METHOD_UNLOAD_START    // issued before unload. Method code is no
                                          // longer being executed, but code and info
                                          // are still in memory. The VTune profiler
                                          // may capture method code and info at
                                          // this point.
} iJIT_JVM_EVENT;



// method

   // An event is occuring in the JIT Compiler
   int WINAPI iJIT_NotifyEvent(
      iJIT_JVM_EVENT event_type,              // the event that occured
      void                *EventSpecificData);  // Data for that event, or NULL if no data


// Data structures for the events.

        typedef   struct _LineNumberInfo {

                unsigned long      Offset;         // Offset from the begining of the method's
                                                   // JITed code.
                unsigned long   BCodeOffset;       // Byte Code Offset
                unsigned long   LineNumber;        // line number or zero if unknown

        } *pLineNumberInfo, LineNumberInfo;
```

```
// JVM_EVENT_TYPE_METHOD_LOAD_FINISHED

typedef   struct _iJIT_Method_Load {
        unsigned long    method_id;                  // uniq method ID
        char              *method_name;              // methode name
        unsigned long    method_load_address;        // virtual address of that method
        unsigned long    method_size;                // Size in memory
        unsigned long    line_number_size;           // Line Table size in number of entries
        pLineNumberInfo  line_number_table;          // Pointer to the beginning of the line numbers info
        unsigned long    class_id;                   // uniq class ID
        char             *class_file_name;           // class file name (Fully qualified)
        char             *source_file_name;          // class source file name (Fully qualified)
} *piJIT_Method_Load, iJIT_Method_Load;


// JVM_EVENT_TYPE_METHOD_UNLOAD_START

typedef   struct _iJIT_Method_Unload {
        unsigned long    method_id;                  // uniq method ID
        unsigned long    class_id;                   // uniq class ID
} *piJIT_Method_Unload, iJIT_Method_Unload;


#ifdef __cplusplus
}
#endif

#endif // __iJITProf_h__
```

For JITed code, we have to have **native_code_offset** and **line_number**. If byte code-to-native code mapping cannot be provided, then **byte_code_offset** can be set to 0, indicating value invalid/not available.

2. ***Symbol Support in JIT***:
This is the next level of support. The mechanism in which this works is very similar to the above. In addition to the level of support given above, for all the variables (local variables/call attributes) referred to in the native code generated, a pointer to a symbol table can be provided in a similar format, using the method id.

3. ***Support in VM/JIT***:

The next, most complete level of support can be made available when both JIT and the VM can get involved. In this scheme, a profile object registers its availability. VM, on checking with the registry entries for the presence of the profile object, creates an instance of it when it loads and initializes the object. The object sets the appropriate event flags for the VM and VM notifies the profile object whenever the event happens.

This is a complete scheme, and the VM and JIT both get involved in notifying the profile object. Here, a lot of information can be made available to the profiler. Events such as Java thread create, class loads/unloads,

interpreted/JITed methods, VM created stub execution, native code execution and everything can be notified. This method is both useful for Java JIT/VM writers and the Java end users.

The following set of routines should be implemented by VM/JIT, with the entries exposed to the VTune profiler:

```
// Event monitors can request notification of these event categories.
typedef enum event_category
{
      CLASS_LOADS             = 0x00000001,
      METHOD_CALLS            = 0x00000002,
      JIT_COMPILATION         = 0x00000004,
      EXCEPTIONS              = 0x00000020,
      MONITOR_OPERATIONS      = 0x00000040,
      GARBAGE_COLLECTIONS     = 0x00000080
}
EVENT_CATEGORY;
typedef enum execution_model
{
      JIT_COMPILED,
      NATIVE,
      INTERPRETED
}
EXECUTION_MODEL;
```

```
DWORD SetEventMask(
    [in] EVENT_CATEGORY events);

  // Get the event notifications requested.

DWORD GetEventMask(
    [in] EVENT_CATEGORY *pevents);

        // Describe a method.

  DWORD MethodInformation(
    [in] MethodID method_id,
    [out] LPSTR *ppmethod_name,
    [out] ClassID *pclass_id,
    [out] EXECUTION_MODEL *pexec,
    [out] int *psource_line_info_length,
    [out] SourceLineInfo **ppsource_line_info);

        // Describe an interpreted method.

  DWORD InterpretedMethodInformation(
    [in] MethodID method_id,
    [out] unsigned int *pbyte_code_length,
    [out] BYTE_CODE **ppbyte_codes);

        // Describe a JIT-compiled method.

  DWORD JITCompiledMethodInformation(
    [in] MethodID method_id,
    [out] unsigned int *pjit_code_length,
    [out] unsigned char **ppjit_code);
```

```
        // Describe a class.

    DWORD ClassInformation(
       [in] ClassID class_id,
       [out] LPSTR *ppclass_file_name,
       [out] LPSTR *ppsource_file_name,
       [out] int *pmethods,
       [out] MethodID **ppmethod_ids,
       [out] __int64 *pobjects_created);

        // Describe an object.

    DWORD ObjectInformation(
       [in] ObjectID objectID,
       [out] ClassID *pclass_id);

        // Get the number of times an object's monitor has been used.

    DWORD GetMonitorUsage(
       [out] __int64 *pmonitor_usage);
```

The following methods are implemented by the profiler DLL with the entries exposed to JIT/VM. JIT/VM will call these routines at appropriate times.

```
// Initialize an event monitor.
    DWORD Initialize(
       [in] LPCSTR pclass_file_name,
       [in] IJavaEventMonitorIDInfo *pmonitor_info,
       [in] DWORD java_flags,              // bit mask of flags from JAVA_STATE_FLAGS
       [out] DWORD *prequested_events);    // bit mask of events from JAVA_EVENT_CATEGORY


// An event from JVM_EVENT_TYPE is about to occur.
    DWORD NotifyEvent(
       [in] JVM_EVENT_TYPE event,
       [in] UniqueID event_id);

    // A method is about to be entered.
    DWORD MethodEntry(
       [in] MethodID method_id,
       [in] StackID stack_id);

    // A method is about to be exited.
    DWORD MethodExit(
       [in] StackID stack_id);

    // The execution of a bytecode instruction is about to occur.
    DWORD ExecuteByteCode(
       [in] MethodID method_id,
       [in] BYTE_CODE *pbyte_code,
       [in] DWORD byte_code_offset);
```

Packaging Logistics

The above APIs specify, in a nutshell, the information required for VTune to work in a Java VM environment. If some of the above can't be made available in a JIT/VM environment, VTune can still work with the minimal set. The details in the third level of support can also be worked out.

For any level of support, there will be a profile object (could also be a DLL) that is required to be in the VM environment. There are multiple ways this can be packaged:

1. Intel can provide the DLL and header files necessary to include them to the VM/JIT (with possibly a source file also of actual DLL invocations). The JIT/VM vendor can include them with their shipment.
2. Intel provides the profiler object with VTune and registers its availability. When VM starts up, it checks for this availability, checks if the profiling needs to be turned on, and then can start loading/using the profile object. The JIT/VM vendor and VTune team need to agree on the exact registry settings for this.

Third party trademarks are the property of their owners.